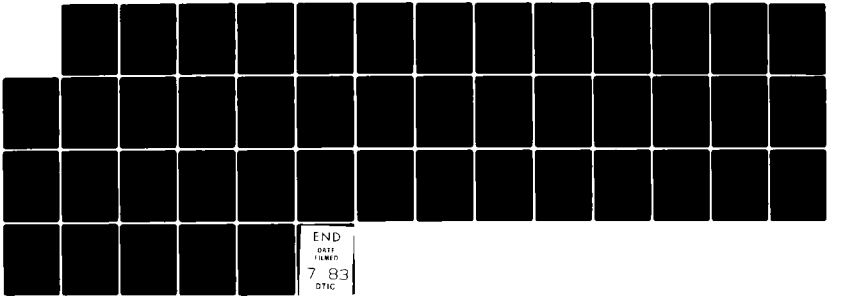
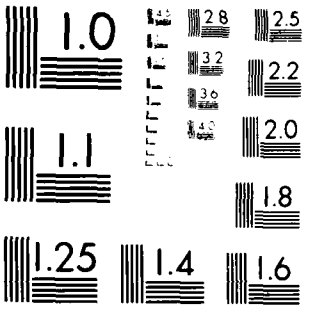
AD-A130 048 RESEARCH ON SYNTHESIS OF CONCURRENT COMPUTING SYSTEMS 1/1
(U) KESTREL INST PALO ALTO CA R M KING ET AL. SEP 82
AFOSR-TR-83-0562 F49620-82-C-0007

UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
7 83
DTIC



KES.U.82.10

# RESEARCH ON SYNTHESIS OF CONCURRENT COMPUTING SYSTEMS



by

Richard M. King

Thomas C. Brown

Cordell Green
Principal Investigator

Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304

September, 1982

FINAL TECHNICAL REPORT

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling AFB, DC 20332

Research sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under contract F49620-82-C-0007. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|
| 1. REPORT NUMBER: AFOSR-TR- 83-0562 | 2. GOVT ACCESSION NO.: AD-A130048 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle): "Research On Synthesis of Concurrent Computing Systems" | 5. TYPE OF REPORT & PERIOD COVERED: FINAL TECHNICAL REPORT 02 OCT. 1981 - 30 SEPT. 1982 |
| | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s): Richard M. King, Cordell Green | 8. CONTRACT OR GRANT NUMBER(s): F49620-82-C-0007 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: Kestrel Institute, 1801 Page Mill Road, Palo Alto, CA 94304 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: AFOSR/NM Bldg 410 Bolling AFB DC 20332 | 12. REPORT DATE: 30 September 1982 |
| | 13. NUMBER OF PAGES: 145 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research, Building 410, Bolling AFB, D.C. 20332 | 15. SECURITY CLASS. (of this report): Unclassified |
| | 15a. DECLASSIFICATION DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Concurrency, Parallelism, Architectures, Synthesis, Transformation,

Interprocessor communication, Connectivity reduction

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We discuss the codification of programming knowledge for the synthesis of concurrent programs. Herein is presented a semiformal derivation of two concurrent algorithms: a concurrent version of a dynamic programming algorithm and concurrent array multiplication. Both derived parallel structures run in linear time. The concurrent versions are significant and complex algorithms, though they are not new and already have been reported in the literature. The synthesis knowledge for these derivations is embodied in seven synthesis rules, preliminary versions of which are

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

BLOCK NO. 20. Abstract (con't).

presented in this report. The rules will probably generalize to other classes of algorithms.

We have also discovered a pair of techniques called *virtualization* and *aggregation*. This pair of techniques (plus the other seven rules) is shown to be powerful enough to synthesize Kung's systolic array architecture from a specification of matrix multiplication.

# Contents

# Plates

# Abstract

The object of our research is the codification of programming knowledge for the synthesis of concurrent programs. This final report presents the derivation of two concurrent algorithms: dynamic programming (for the class of problems that run in polynomial time on sequential machines) and array multiplication. Both derived concurrent versions run in linear time. The concurrent versions are significant and complex algorithms, though they are not new and already have been reported in the literature. The synthesis knowledge for these derivations is embodied in seven synthesis rules, preliminary versions of which are presented in this report. The rules will probably generalize to other classes of algorithms but we have not explored that issue yet.

We have also discovered a pair of techniques called *virtualization* and *aggregation*. This pair of techniques (plus the other seven rules) is shown to be powerful enough to synthesize Kung's systolic array architecture [Kung-76] from a specification of matrix multiplication.

# Introduction

In this paper we describe methods for synthesizing parallel structures from concise, very high level specifications of algorithms. We use the very high level language, V, which can express both programs and program transformation rules. In order to allow for reasoning about concurrency, we have defined language constructs to express parallel structures that solve a class of problems. In these problems the number of processors is a nonconstant polynomial in some measure of the problem size. We then developed rules, or abstract input/output specifications, that transform specifications of sequential algorithms written in V into parallel structures that accomplish the same tasks. We have coded some of them in V.

First, rules operate on specifications by identifying processing that can be performed concurrently on distinct elements of arrays that describe either the problem, its solution, or some intermediate results. They then add specifications of multiple processors, each with responsibility for a portion of the input data, and a specification of the interactions among the processors. Next, other rules reduce the degree of interconnection between the processors whenever that degree is not asymptotically constant but is polynomial in the size of the problem. We apply these rules to a subclass of dynamic programming specifications and to a specification of matrix multiplication, and have derived asymptotically fast, sparsely interconnected networks.

We have also developed techniques to create Kung's systolic array parallel structure from a specification of matrix multiplication. We have identified and formalized a powerful pair of techniques, which we call *virtualization* and *aggregation*, for producing certain parallel structures that are often complex (and generally recognized as "clever"), given only high level specifications.

Intuitively, virtualization is the addition of one or more dimensions to an array, turning each single element into a column (or plane or hyperplane) that contains the partial results of the computation of that element. For example, if $A_{i,j}$ is computed using a single enumeration, then virtualization would produce a three dimensional array, say $A'$, and $A'_{i,j,k}$ would contain the $k^{\text{th}}$ partial result of this enumeration. Virtualizations we have studied reduce the computation per array element to $O(1)$.

Also intuitively, aggregation is the grouping together of processors, each of which does a small amount of work, into groups of processors, each represented by a single processor. Each processor does all of the work that any processor in its original group did, but this can still be done quickly because each of the processors in the original group had a small amount of work to do, and no two processors had to do their work at overlapping times. There exist an enormous number of ways to group processors, but we will use only simple ones.

# Mathematical Notation

| | |
|---|---|
| $P$ | number of processors used |
| $\vec{B}$ or $\vec{B}_j$ | (where $B$ is any letter) a vector of $b_i, l \leq i \leq u$ for some lower limit $l$ and upper limit $u$. Where $l$ and $u$ are particularly important and non-obvious, $\langle b_l, \ldots, b_u \rangle$ (or $\langle b_l \rangle$ where $l$==$u$) may be used. |
| $\vec{I} \| \vec{J}$ | the concatenation of the vectors, $\vec{I}$ and $\vec{J}$ |
| $\lvert\vec{I}\rvert$ | the length of the vector $\vec{I}$ |
| $\langle\langle l \ldots u \rangle\rangle$ | the *ordered* sequence (not set) of integers from $l$ to $u$, inclusive |
| $n$ | $n$ will *always* be used to denote some measure of the size of a problem to be solved by an algorithm or a parallel structure. |
| #⟨set⟩ | cardinality of the set |
| $\theta(g(n))$ | *Order* $g(n)$ where precisely known. This means that $g(n)$ is (within a constant factor) the best estimate of whatever is being measured as n increases. Formally, $f(n)$=$\theta(g(n))$ is defined as<br><br>$\exists$ constants $c$, $c'$, $c''$ *where* $n > c \Rightarrow c'g(n) \leq f(x) \leq c''g(n)$<br><br>$f(n)$ is called the *asymptotic behavior* of $g$ or $g(n)$ |

Section 1

# Problem Description, Solution Techniques and Rules

by
Richard M. King
Kestrel Institute
October 1982

We have been studying the derivation of parallel computation structures which achieve an asymptotic improvement in the computation time, as compared with the best known sequential algorithm. To achieve this the number of processors in use must grow with the size of the problem. We will be interested in cases where $P \geq \theta(n)$, because these offer the greatest opportunities for sharing the work among a large number of processors.

Algorithms whose asymptotic running time is $\theta(n^i)$ for $i > 1$ often use an internal aggregated data structure whose size is $\theta(n^j)$ for some $1 \leq j \leq i$. We try to create parallelism by assigning a processor to each element of the aggregated data structure. The structures most important in this work are sets, arrays of various dimensionality, and stacks. This paper considers arrays. Data structure selection for an algorithm dependent on stacks or sets can produce arrays, so this choice is not overly restrictive or unnatural. Another important issue in parallel algorithm synthesis is the connectivity of the resulting multiprocessor net. This is especially important because we seek asymptotic growth in the number of processors, so too rich a connectivity may result in a collection of processors and interconnections that would be impossible to fabricate economically. We thus give attention to reducing connectivity.

In this paper the term *parallel structure*, or simply *structure*, will be used to denote a program designed for a $\theta(n)$ or larger collection of processors plus a specification of how they should be interconnected.

## §1.1 Taxonomy of the Synthesis Task

Figure 1 is a taxonomy of the various states that a synthesis process can be in, together with the possible synthesis steps. We will use such phrases as "a Class D synthesis" throughout this document. In the taxonomy, structures to the right are more desirable than the ones on the left, because they require fewer connections between processors. Each labelled arc represents a possible synthesis step.

It might seem that every Class D synthesis (for example) is harder than any Class A or Class B synthesis, since the result of a Class D synthesis is the same as the result of a Class A followed by a Class B synthesis. This is not true in general, although it usually holds. Some specifications are especially suitable for some of the "higher" syntheses. One example is that a specification including backtracking is often more easily synthesized into a tree-structured parallel structure than any other.

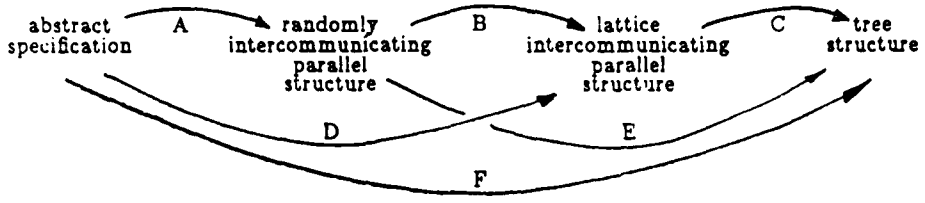



*Figure 1.* Taxonomy of Syntheses

We concentrate on Class D synthesis in this report because it represents an advance on our prior work on Class A synthesis ([GCP-81]).

## §1.2 A Case Study: Polynomial-Time Dynamic Programming

We have examined a class of Polynomial time ($P$-time) dynamic programming algorithms for which it is possible to synthesize an optimal parallel scheme. The synthesis uses rules displayed in § 1.3, and inference capabilities which Kestrel proposes to develop in 1983, described in [Brown-82]. Abstractly programmed algorithms in this class include the Cocke-Younger-Kasami parsing algorithm for a fixed, possibly ambiguous Chomsky Normal Form grammer, described in [AhoUll-72]; the Optimal Binary Search Tree algorithm, described in [Knuth-73]; and Optimal Multiple Matrix Multiplication, described in [AHU-74]. All of the algorithms fit into the following scheme.

Each algorithm generates the "solution" to a problem whose input is a sequence $\vec{S}$ of $n$ items by using a dynamic programming technique. This technique generates a solution for a sequence of items by combining solutions for contiguous subsequences. The solution $V(\vec{K})$ for a sequence $\vec{K}$ of length $m$ is found by:

1. Generating the $m-1$ possible partitions of $\vec{K}$ into contiguous subsequences $\vec{I}$ and $\vec{J}$ such that $\vec{I}\|\vec{J}=\vec{K}$;
2. Forming for each partition a partial solution for $\vec{I}\|\vec{J}$ by applying a function $F$ to $V(\vec{I})$ and $V(\vec{J})$;
3. Obtaining $V(\vec{I}\|\vec{J})$ by combining (using a binary operation $\odot$) all of the partial solutions. This is expressed formally below:

$$V(\vec{K}) = \bigodot_{\vec{I},\vec{J}:\vec{I}\|\vec{J}=\vec{K}} F(V(\vec{I}), V(\vec{J}))$$

In order to obtain the following parallel structure that runs in time $\theta(n)$, two conditions must hold:

- Both $\odot(x,y)$ and $F(x,y)$ must take constant time,
- $\odot$ must be both commutative and associative. This allows $F(V(\vec{I}), V(\vec{J}))$ values to be included in the running $\odot$-total in any order they become available.

These conditions are met by a sizable class of algorithms, e.g. the algorithms mentioned above. The algorithm generates the solution $V(\vec{S})$ for the original problem $\vec{S}$ of length $n$. The process starts with $V((s_i))$ for each $s_i \in \vec{S}$, then generates solutions for subsequences of length 2, 3, and so on, up to $n$. We give below two dynamic programming algorithms that fit into this scheme.

The Cocke-Younger-Kasami algorithm parses a sequence of terminal symbols according to a fixed grammar, $G$, in Chomsky Normal Form. This form specifies that each production rule in the grammar is either of the form $N \rightarrow t$ for nonterminal $N$ and terminal $t$, or $N \rightarrow PQ$ for nonterminals $N$, $P$, and $Q$. In this parsing algorithm, each problem is a sequence of terminal symbols, $\vec{T}$, and the solution $V(\vec{T})$ is the set of nonterminal symbols that derive $\vec{T}$. Let the initial terminal sequence be $(t_1 \ldots t_n)$. Then $V((t_i))$ are those nonterminals $N$ for which there is a production rule in the grammar of the form $N \rightarrow t_i$. Given two sequences of terminals $\vec{A}$ and $\vec{B}$, the nonterminals that

produce $\vec{A}\|\vec{B}$ include those nonterminals $N$ for which there is a rule $N \rightarrow PQ$ where $P \in V(\vec{A})$ and $Q \in V(\vec{B})$. The nonterminals that produce a sequence $\vec{S}$ are obtained by dividing the sequence $\vec{S}$ into two subsequences in all possible ways and taking the union of the results. In our formalism,

$$F(V(\vec{S}), V(\vec{T})) = \{N | [N \rightarrow PQ] \in G \wedge P \in V(\vec{S}) \wedge Q \in V(\vec{T})\}$$

and

$\odot$ is the Union operation, which is indeed associative and commutative.

Another example of a dynamic programming algorithm fitting our scheme is finding the complexity of the optimal grouping to multiply a given sequence $(M_1, M_2 \ldots M_n)$ of matrices. Since matrix multiplication is associative, multiplying the matrices in different groupings produces the same result matrix, but different groupings may have different execution efficiencies. If $M$ is a $p \times q$ matrix, and $N$ is a $q \times r$ matrix, then the product $M \times N$ will be a $p \times r$ matrix, and the multiplication will execute in time proportional to $pqr$ (if a simple matrix multiplication algorithm is used).

This problem fits into the scheme presented above in the following fashion. The "solution" for each matrix subsequence $V((M_i \ldots M_j))$ is a triple $(p, q, c)$: $p$ is the row size of $M_i$; $q$ the column size of $M_j$ (since multiplication using any grouping of $(M_i \ldots M_j)$ results in a $p \times q$ matrix) and $c$ is the optimal execution cost for computing $M_i \times \cdots \times M_j$. The $F$ for this algorithm is defined below:

$$F((p_1, q_1, c_1), (p_2, q_2, c_2)) \equiv (p_1, q_2, c_1 + c_2 + p_1 q_1 q_2)$$

$\odot$ for this algorithm returns the triple with the minimum cost element. (Since only the costs can differ among triples, $\odot$'s choice is arbitrary if the costs happen to be the same.) The *minimum* operation is associative and commutative.

A high-level specification of the dynamic programming algorithm is presented below. A subsequence can be represented by its length and where it begins. The array $A$ used below contains solutions to subsequences: the element $A_{l,m}$ contains $V((i_l, \ldots, i_{l+m-1}))$, where $\vec{I}$ is the initial sequence. The complexity of each "executable" statement is presented at the right.

The algorithm specification is as follows:

| | |
|---|---|
| **ARRAY** $A_{l,m}, 1 \leq m \leq n, 1 \leq l \leq n-m+1$ | |
| **INPUT ARRAY** $v_l, 1 \leq l \leq n$ | |
| **ENUMERATE** $l \in (\!(1 \ldots n)\!)$ **do** | $\theta(1)$ |
| $A_{l,1} \leftarrow v_l$ | $\theta(n)$ |
| **ENUMERATE** $m \in (\!(2 \ldots n)\!)$ **do** | $\theta(1)$ |
| **ENUMERATE** $l \in \{1 \ldots n-m+1\}$ **do** | $\theta(n)$ |
| $A_{l,m} \leftarrow \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$ | $\theta(n^3)$ |

*Figure 2.* Specification of $\theta(n^3)$ Dynamic Programming

A cost of $\theta(n^3)$ is assigned to the evaluations of $F$ and $\odot$ because it is given that a single evaluation of both $F$ and $\odot$ takes constant time.

The time complexity of the specified algorithm executed on a sequential machine is indeed $\theta(n^3)$.* However, it is possible to implement the specification on a two-dimensional array of $\theta(n^2)$ processors and the resulting algorithm will run in $\theta(n)$ time. The memory size of each processor is $\theta(n)$. Below

*A trick is available for Optimal Binary Search Tree. This trick involves bounding $k$ in Figure 2 more narrowly than $\{1 \ldots m-1\}$. This trick reduces the algorithm's running time to $\theta(n^2)$, but it does not generalize to the other algorithms. We know of no analog to this trick for parallel structures.

we describe the operation of the structure, and then prove that it is a $\theta(n)$ algorithm. This algorithm has been reported in the literature [GKT-79].

The network of processors is displayed in Figure 3. Observe that $P_{l,m}$ is connected to $P_{l,m-1}$ and $P_{l-1,m-1}$. Each processor $P_{l,m}$ will compute the value of $A_{l,m}$. To do this it needs two streams of information: $A_{l,k}$ and $A_{l+k,m-k}$, where $k < m$. These streams of data come respectively over wires from processors $P_{l,m-1}$ and $P_{l+1,m-1}$. Each processor $P_{l,m}$ (except $P_{1,n}$) will send every $A$-value received from $P_{l,m-1}$ to $P_{l,m+1}$ and from $P_{l+1,m-1}$ to $P_{l-1,m+1}$ as soon as $P_{l,m}$ gets it. Each processor will also compute $F$-values and merge them into a running $\odot$-total as soon as it gets the $A$-values necessary.

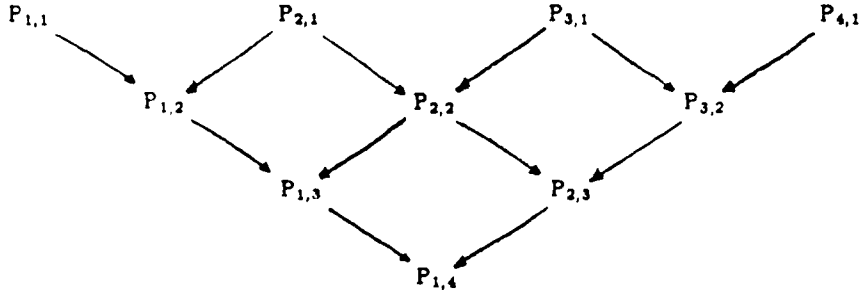



*Figure 3.* Processor Interconnections

At first glance, it might appear that this algorithm has time complexity $\theta(n^2)$. Each processor needs to receive $\theta(n)$ $A$-values from each of its incoming wires; it must at some time perform $\theta(n)$ worth of computation on the data received before it sends its result on each of its outgoing wires. However, a careful timing argument shows that an execution time of $\theta(n)$ can be achieved.

**Definition 1.1.** *Within $P_{l,m}$, for any $k$ where $1 \leq k < m$, $A_{l,k}$ and $A_{l+k,m-k}$ are called a* complementary pair *of $A$-values.*

Processor $P_{l,m}$ will apply $F$ to each complementary pair of $A$-values.

The next lemma shows that each processor $P_{l,m}$ receives all $2m-2$ values it needs, though it waits $\theta(m)$ for its first complementary pair, $A_{l,\lceil m/2 \rceil}$ and $A_{l+\lceil m/2 \rceil, m-\lceil m/2 \rceil}$.

**Lemma 1.2.** *Each processor $P_{l,m}$ where $1 \leq m > 1$ receives the values $A_{l,m'}$ where $1 \leq m' < m$ and (separately) $A_{l+m-m',m'}$ where $m' < m$ in order of increasing $m'$.*

*Proof:* By induction on $m$. Clearly this is true for $P_{l,2}$, which receives only one value on each of its incoming wires. Now suppose it is true for $P_{l,m-1}$ and $P_{l+1,m-1}$. Then $P_{l,m}$ will receive $A$-values in the proper order from $P_{l,m-1}$ and $P_{l+1,m-1}$ through $m'=m-2$, following which it receives $A_{l,m-1}$ and $A_{l+1,m-1}$ from those processors. But the latter two $A$-elements are just those required to preserve the sequences. ∎

system startup $T=0$, and after $z$ units of time $T=z$. The time unit satisfies the first condition of the following lemma.

**Lemma 1.3.** *If all of the following conditions are met:*

- *All of the following takes processor $P_{l,m}$ no more than one unit of time: receiving two values, one each from $P_{l,m-1}$ and $P_{l+1,m-1}$; sending these values on to $P_{l,m+1}$ and $P_{l-1,m+1}$; applying the function $F$ twice to two complementary pairs of $A$-values if all values are available; and merging the resulting value into a running $\odot$-total.*
- *The $A$-values come into $P_{l,m}$ in the order indicated by Lemma 1.2.*

- *Each processor* $P_{l,m}$ *sends values received from* $P_{l,m-1}$ *resp.* $P_{l+1,m-1}$ *to* $P_{l,m+1}$ *resp.* $P_{l-1,m+1}$ *no later than one time unit after receipt.*
- *At* $T=0$ *processor* $P_{l,1}$ *transmits* $A_{l,1}$.

*then* $P_{l,m}$ *will compute* $A_{l,m}$ *no later than* $T=2m$.

*Proof*: By induction $P_{l,1}$ is initialized to know $A_{l,1}$. Now suppose the lemma is true for $m' < m$ and suppose $T=2m''$ for some $m'' < m$. First prove a sublemma, that at this time $P_{l,m}$ will have included at least $\max(0, 2(m''-\lceil m/2\rceil))$ $F$-values in its running $\odot$-total. This sublemma is proven by induction on $m''-\lceil m/2\rceil$.

When reading the proof of the sublemma, keep in mind that the "life" of a processor $P_{l,m}$ is divided into three epochs:

1. When $T < m$, the processor may have received no $A$-values.
2. When $m \leq T < 1\frac{1}{2}m$, the processor will have received at least $T-m$ $A$-values from each of its input lines. Since the first half of the $A$-values from each inbound wire form complementary pairs with the *last* half of the values from the other inbound wire, $P_{l,m}$ may not have been able to perform any calculations of any $F$-values yet.
3. When $T \geq \frac{1}{2}m$, the processor will have received at least half (more accurately, at least $T-m$) of the values from each inbound wire. During each unit interval, it will receive one $A$-value from each inbound wire, which will "match" with some value that was stored from the other wire during epoch 2. Two $F$-calculations will be possible – one for each of the just-received inbound data.

If $m''-\lceil m/2\rceil=0$ the sublemma requires nothing. If $m''-\lceil m/2\rceil > 0$, consider the situation $m''-\lceil m/2\rceil$ before $T=2m''$. All processors $P_{l,j}$ and $P_{l+j,j}$, where $j \leq \lceil m/2\rceil + m''-\lceil m/2\rceil$ will have completed their work and their answers will have had time to reach $P_{l,m}$. Thus at least $2(m''-\lceil m/2\rceil)$ pairs of $A$-values will have arrived. Since (by induction on $m''-\lceil m/2\rceil$) two time units ago $2(m''-\lceil m/2\rceil)-2$ $F$-values had already been merged into the running $\odot$-total there is plenty of time to merge two new $F$-values into the running $\odot$-total, completing the induction step of the sublemma.

Lemma 1.3 follows immediately from the sublemma and the observation that the merging of $m-1$ $F$-values into the running $\odot$-total in $P_{l,m}$ constitutes a calculation of $A_{l,m}$. ∎

**Theorem 1.4.** *The time to compute* $A_{1,n}$ *is* $\theta(n)$.

*Proof*: Immediate from Lemma 1.3. ∎

In the next section we will show how this parallel structure can be derived from the specification in Figure 2.

## §1.3 Rules for Parallel Structure Synthesis

Rules for the Class A synthesis task appear elsewhere (see [GCP-81]). This report describes the task of synthesis of parallel structures for arrays of processors in which the interconnections describe a k-dimensional lattice for some k, i.e. a Class D synthesis task.

As examples of rule application and a demonstration of the rules' effectiveness, we apply each rule to the $P$-time dynamic programming specifications. We will repeat that specification here, augmented

with output array descriptions.

$(P.1)$ **ARRAY** $A_{l,m}, 1 \leq m \leq n, 1 \leq l \leq n-m+1$
**INPUT ARRAY** $v_l, 1 \leq l \leq n$
**OUTPUT ARRAY** $O$
**ENUMERATE** $l \in \langle\langle 1 \ldots n \rangle\rangle$ **do** $\theta(1)$
  $A_{l,1} \leftarrow v_l$ $\theta(n)$
**ENUMERATE** $m \in \langle\langle 2 \ldots n \rangle\rangle$ **do** $\theta(1)$
  **ENUMERATE** $l \in \{1 \ldots n-m+1\}$ **do** $\theta(n)$
    $A_{l,m} \leftarrow \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$ $\theta(n^3)$
$O \leftarrow A_{1,n}$ $\theta(1)$

*Figure 4.* Specification of $\theta(n^3)$ Dynamic Programming with Explicit I/O

### 1.3.1 Preparatory Rules

The problems now amenable to Class D synthesis have internal arrays of storage, and the requirement is to fill in the array by computing a value for each element. Our strategy will be to assign a processor t . each element of the array. This rule declares a processor for each element of the main array in the problem, and composes a single enumerated **PROCESSORS** statement. This statement has several clauses: the processors definition clause, the **HAS** clause, the **HEARS** clause(s), and the **USES** clause(s). Any part of the **PROCESSORS** statement except the processors definition clause can be made conditional. An example of a **PROCESSORS** statement is shown below:

**PROCESSORS** $P_{l,m}, 1 \leq m \leq n, 1 \leq l \leq n-m+1$
  **HAS** $A_{l,m}$
  **if** $m=1$ **then USES** $v_l$, **HEARS** Q
  **if** $2 \leq m \leq n$ **then**
    **USES** $A_{l,k}, 1 \leq k \leq m$
    **USES** $A_{l+k,m-k}, 1 \leq k \leq m$
    **HEARS** $P_{l,m-1}$
    **HEARS** $P_{l+1,m-1}$

This statement means all of the following:

- A family of processors exists. The family name is P. Each member of the family is named by two indices, and any member $P_{l,m}$ exists if $1 \leq m \leq n \wedge 1 \leq l \leq n-m+1$. The value $n$ is an externally defined constant value (for any instance of the problem) defining the problem size. This **PROCESSORS** statement actually declares some facts about every processor in the family.
- Each element, $P_{l,m}$, of this family is responsible for computing the value of (i.e. **HAS**) $A_{l,m}$. $A$ is an array declared elsewhere in the specification that contains the **PROCESSORS** statement.
- If $P_{l,1}$ is defined it needs $v_l$ to compute its **HAS** values, and it expects to get these values from (i.e. **HEARS**) the (only) processor in the Q family.
- If $P_{l,m}$ is defined and $2 \leq m \leq n$, then $P_{l,m}$ needs the values of $A_{l,k}$ for any $k$, $1 \leq k \leq m-1$. It also needs $A_{l+k,m-k}$ for any $k$ in that range. It expects to get these values from processors in the P family, namely $P_{l,m-1}$ and $P_{l+1,m-1}$. The scope of the bound variables list (in this case, "$l, m$") is the entire **PROCESSORS** statement. Two **PROCESSORS** statements must have distinct processor names (in this case, "P").

1.3.1.1 Rule A1: Give Each Non-I/O Array Element its Own Processor

By our conventions, the portion before the " → " is the *antecedent* and the rest is the *consequent*. Variables free in the antecedent are implicitly existentially quantified and the scope of this quantification is the entire rule. Variables free only in the consequent are universally quantified (but this is rare). A rule is said to *apply* if the antecedent is true; when this happens the semantics of the rule is to make the consequent true. It is explicitly permissible for the consequent to make the antecedent no longer true.

*rule* *MAKE-PSs* (**) TRANSFORM
  *X* STATEMENT
  ∧ *X* ∈ ** STATEMENTS
  ∧ *X*:'ARRAY $NAME_{BOUND}$ *ENUMERS*'
  ∧ *Y*=(GENSYM 'PROC)
  ∧ *Z*:'PROCESSORS $Y_{BOUND}$ *ENUMERS* HAS $NAME_{BOUND}$'
→
  *Z* ∈ ** STATEMENTS

of this quantification is the entire rule. Variables free only in the

*MAKE-PSs* applied to (*P*.1) binds as follows:

bindings:

**=(*(entire specification)*)
*X*='ARRAY $A_{l,m}, 1 \le m \le n, 1 \le l \le n-m+1$'
*NAME*='A'
*BOUND*='$l, m$'
*ENUMERS*='$1 \le m \le n, 1 \le l \le n-m+1$'
*Y*='P'
*Z*='PROCESSORS $P_{l,m}, 1 \le m \le n, 1 \le l \le n-m+1$
HAS $A_{l,m}$'

obtaining

(*P*.2)

ARRAY $A_{l,m}, 1 \le m \le n, 1 \le l \le n-m+1$
PROCESSORS $P_{l,m}, 1 \le m \le n, 1 \le l \le n-m+1$ HAS $A_{l,m}$
INPUT ARRAY $v_l, 1 \le l \le n$
OUTPUT ARRAY *O*
ENUMERATE $l \in \langle\langle 1 \ldots n \rangle\rangle$ do — $\theta(1)$
  $A_{l,1} \leftarrow v_l$ — $\theta(n)$
ENUMERATE $m \in \langle\langle 2 \ldots n \rangle\rangle$ do — $\theta(1)$
  ENUMERATE $l \in \{1 \ldots n-m+1\}$ do — $\theta(n)$
    $A_{l,m} \leftarrow \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$ — $\theta(n^3)$
O ← $A_{1,n}$ — $\theta(1)$

as the new state of the database.

#### 1.3.1.2 Rule A2: Assign I/O Arrays to Processors

This rule assigns a *single* processor to each input or output array. The reason only a single processor is assigned is that it is assumed that input values will reside in a single entity, such as a tape drive.

*rule* *MAKE-IOPSs* (**) TRANSFORM
$X$.STATEMENT
$\wedge\ X \in$ **.STATEMENTS
$\wedge\ X$:'*IO* ARRAY $NAME_{BOUND}$ *ENUMERS*'
$\wedge$ (*IO*='INPUT $\vee$ *IO*='OUTPUT )
$\wedge\ Y$=(GENSYM 'PROC )
$\wedge\ Z$:'PROCESSORS $Y$ HAS $NAME_{BOUND}$ *ENUMERS*'
$\rightarrow$
$Z \in$ **.STATEMENTS

Rules *MAKE-PSs* and *MAKE-IOPSs* make PROCESSORS statements that do not have USES and HEARS clauses yet. The next rule fills in those clauses, and subsequent rules improve them.

Rule *MAKE-IOPSs* applies for two sets of bindings:

| | |
|---|---|
| **=⟨(*entire specification*)⟩ | **=⟨(*entire specification*)⟩ |
| $X$='OUTPUT ARRAY $O$' | $X$='INPUT ARRAY $v_l, 1 \le l \le n$' |
| *IO*='OUTPUT | *IO*='INPUT |
| *NAME*=$O$ | *NAME*=$v$ |
| *BOUND*=(*empty string*) | *BOUND*='$l$' |
| *ENUMERS*=(*empty string*) | *ENUMERS*='$1 \le l \le n$' |
| $Y$=R | $Y$=Q |
| $Z$='PROCESSORS R HAS $O$' | $Z$='PROCESSORS R HAS $v_l, 1 \le l \le n$' |

resulting in

| | | |
|---|---|---|
| ($P$.3) | ARRAY $A_{l,m}, 1 \le m \le n, 1 \le l \le n-m+1$ | |
| | PROCESSORS $P_{l,m}, 1 \le m \le n, 1 \le l \le n-m+1$ HAS $A_{l,m}$ | |
| | INPUT ARRAY $v_l, 1 \le l \le n$ | |
| \| | PROCESSORS Q HAS $v_l, 1 \le l \le n$ | |
| | OUTPUT ARRAY $O$ | |
| \| | PROCESSORS R HAS $O$ | |
| | ENUMERATE $l \in \langle(1 \ldots n)\rangle$ do | $\theta(1)$ |
| ($P$.3a) | $A_{l,1} \leftarrow v_l$ | $\theta(n)$ |
| | ENUMERATE $m \in \langle(2 \ldots n)\rangle$ do | $\theta(1)$ |
| | ENUMERATE $l \in \{1 \ldots n-m+1\}$ do | $\theta(n)$ |
| ($P$.3b) | $A_{l,m} \leftarrow \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$ | $\theta(n^3)$ |
| ($P$.3c) | $O \leftarrow A_{1,n}$ | $\theta(1)$ |

So far, all rule application can be done in a straightforward manner, without inference.

1.3.1.3 Rule A3: Determine Processors' Inputs

We need rules to describe the connections between processors and the data that processors need to produce results. This rule is very conservative - it determines what array values each processor $P^*$ needs, and it specifies a direct connection from the processors holding those values to $P^*$. The **USES** clause describes the *values* that a processor needs; the **HEARS** clause describes the *processors* that have (**HAS**) these values.

To determine this, consider the innermost loop which assigns values to array elements indexed by non-region-constants. Note that the form of the rule shown below evidences a need for elaborate flow analysis. Non-constant array index expressions are used as processor indices. The indices for those array elements whose values can affect the assigned value comprise the index expressions for the **USES** and **HEARS** sets. A reference at the same loop level will normally generate **USES** and **HEARS** clauses with null enumerations. A reference contained in a deeper loop will normally generate instances of such clauses with inherited enumerators from the loops.

*rule* *MAKE-USES-HEARS* (**) TRANSFORM
**:'PROCESSORS $PDCL_{PBV}$ *PENUMER* HAS $ANAME_{BINDEX}$'
∧ *CB*=**.CONTAINING-BLOCK
∧ *X*=(INNER-LOOP-THAT-DEFINES *ANAME CB*)
∧ *Y* ∈ (ARRAY-REFERENCES-AFFECTING *X*)
∧ *Z*=(EFFECTIVE-ENUMERATOR-OF *Y X*)
∧ *W*.CONDITIONS =*CB*.CONDITIONS ∪(INFERRED-CONDITIONS *X*)
∧ *W*.CLASS =USES-CLAUSE
∧ *W*.ARG ='*ANAME*
(REL-BV *PBV X*.DEF-OF .INDEX-EXPR *Y Z*)
(RELENUMER *PBV X*.DEF-OF .INDEX-EXPR *Y Z*)'
∧ *Q*.CONDITIONS =*CB*.CONDITIONS ∪(INFERRED-CONDITIONS *X*)
∧ *Q*.CLASS =HEARS-CLAUSE
∧ *HISBV*=*ANAME*.PROCSTMT .PROC-BV-OF
∧ *Q*.ARG ='*ANAME*.PROC-OF
(REL-BV *HISBV*
*X*.DEF-OF .INDEX-EXPR *Y Z*)
(RELENUMER *HISBV X*.DEF-OF .INDEX-EXPR *Y Z*)'
→
*W* ∈ **.clauses
∧ *Q* ∈ **.clauses

The **INNER-LOOP-THAT-DEFINES** function finds an innermost locality where an element from the argument array is defined (not merely used). The **ARRAY-REFERENCES-AFFECTING** function returns a set of all points in the program where an array is referenced and the value returned can affect the results of its operand, a program point. The **EFFECTIVE-ENUMERATOR-OF** function determines what (possibly implicit) enumerators its first argument (an array reference) is controlled by, beyond the enumerators that control its second argument (an array definition in this case).

The map, *z*.CONDITIONS, allows any node *z* to be placed under the influence of conditions (an **if** clause). **INFERRED-CONDITIONS** is a function that produces an **if** clause that specifies exactly those conditions that must be true for the point representing the argument to be reached (a form of assertion propagation).

**REL-BV** and **RELENUMER** give a piece of text that respectively will serve as a bound variable and an enumerator for the fragment enumerated by the fourth argument to be valid for the third

argument in the context of the second argument, using the bound variables of the first argument. This would be the bound variables of the fourth argument unless there is a variable name clash.

This modifies the first **PROCESSORS** statement, which becomes

**PROCESSORS** $P_{l,m}, 1 \leq m \leq n, 1 \leq l \leq n-m+1$
**HAS** $A_{l,m}$
| **if** $m=1$ **then USES** $v_l$, **HEARS** Q

Application to the assignment to $A_{l,m}$ in ($P.3b$) produces

**PROCESSORS** $P_{l,m}, 1 \leq m \leq n, 1 \leq l \leq n-m+1$
**HAS** $A_{l,m}$
**if** $m=1$ **then USES** $v_l$, **HEARS** Q
| **if** $2 \leq m \leq n$ **then**
| **USES** $A_{l,k}, 1 \leq k \leq m$
| **USES** $A_{l+k,m-k}, 1 \leq k \leq m$
| **HEARS** $P_{l,k}, 1 \leq k \leq m$
| **HEARS** $P_{l+k,m-k}, 1 \leq k \leq m$

Finally, apply *MAKE-USES-HEARS* one last time, to the null "enumeration", ($P.3c$), that sends the output value to the output "array", $O$. This forces us to modify R's **PROCESSORS** statement as follows:

**PROCESSORS** R **HAS** $O$
| **USES** $A_{1,n}$ **HEARS** $P_{1,n}$

This statement is in its final form.

The applications of *MAKE-USES-HEARS* require flow analysis and some ability to reason about enumeration (to construct **if** clauses).

### 1.3.2 Optimization Rules

The rest of the rules described in this section will transform the simplest parallel structures into more efficient ones. They do this by detecting and removing redundant interconnections.

#### 1.3.2.1 Rule A4: Improve HEARS clauses

It may be that a **HEARS** clause of a **PROCESSORS** statement requires each processor to be connected to more than one other processor. This is undesirable, because the number of interconnections in the whole collection of processors would grow faster than the number of processors, and the cost of interconnections would exceed the cost of processors for sufficiently large problems. This would, in turn, decrease the size of the largest problem that could be handled by a given parallel structure.

However, often it is not necessary for each processor to be connected to all other processors whose values it needs. If processor $P_a$ needs values from processors $P_b$ and $P_c$, but $P_b$ needs a value from processor $P_c$, it may not be necessary for $P_a$ to be connected to $P_c$. $P_a$ must be connected to $P_b$, but $P_b$ will be able to get the value that $P_a$ wants from $P_c$, so it ($P_b$) can pass that datum along.

This form of this observation only secures a constant factor reduction in the number of interconnections (in this case, from two to one), but it is possible to do better by extending the principle. Suppose, for example, that a structure includes a family of processors $P_i$ for $1 \leq i \leq n$. Further suppose that $\forall i, j$ where $j < i$, $P_i$ needs values from $P_j$. In this case, $P_{j+1}$ will need all the values $P_i$ needs, plus the value in $P_j$ itself.

**Basic Observation 1.5.** *In a case such as this $P_j$ is capable of supplying all of the information that $P_{j+1}$ needs, so it is possible to modify the structure to replace the $\theta(n)$ connections required by this* HEARS *clause by a single connection.*

**Definition 1.6.** *In a parallel structure, a* family *of processors is the set of processors defined by a single* PROCESSORS *statement when enumerated over the* PROCESSORS *clause's enumerator. That family is* generated *by that* PROCESSORS *statement.*

**Definition 1.7.** *The set of processors in a processor $P_a$'s family* HEARd *by $P_a$ due to a* HEARS *clause $H_0$ will be written $H_0(P_a)$.*

**Definition 1.8.** *Consider $H_0(P_a)$ and $H_0(P_b)$. Suppose that each is a subset of the same family as $P_a$ and $P_b$ (which are in the same family because they both have the same* HEARS *clause, $H_0$). The interconnections defined by $H_0$* telescope *if these sets $H_0(P_a)$ and $H_0(P_b)$ either are disjoint or one strictly contains the other, for any choice of $P_a$ and $P_b$ in the family. We also say that $H_0$* telescopes. *If $\forall_{P_a, P_b \in family} : [\emptyset \subset H_0(P_a) \subset H_0(P_b) \Rightarrow \exists_{P_c \in family} : [H_0(P_a) \bigcup \{P_a\} = H_0(P_c)]]$ then $H_0$* snowballs. *The notion of a* USES *clause telescoping is defined similarly. A partition is* induced *by a telescoping clause $c_0$ if two processors are in the same partition whenever the sets defined by $c_0$ overlap.*

**Theorem 1.9.** *If a* HEARS *clause $H_0$ snowballs, it can be replaced by another* HEARS *clause that only specifies input from a single processor.*

*Proof*: Consider the family of processors described by the PROCESSORS statement that contains the HEARS clause. Consider also the induced partition $\Pi$.

If the cardinality of an equivalence class $E \in \Pi$ is (say) $c$, then $\forall P_z \in E : |H_0(P_z)| < c$. (No processor can HEAR itself because it would never be able to complete its calculation if it needed its own result to do so.) Since $\forall x, y : x \neq y \Rightarrow |H_0(P_x)| \neq |H_0(P_y)|$, and since $|\{0 \ldots c-1\}| = c$, the processors in $E$ can be completely ordered by the cardinalities of their HEARd sets. By the basic observation and the snowballing property, each processor can get the information that $H_0$ requires from the processor that is its predecessor in this ordering. ∎

**Definition 1.10.** *We call replacing a* HEARS *clause, as in the previous theorem,* reducing *the clause.*

We expect to be able to prove the following result; it "falls out" of a generalization of Theorem 1.5 for which we are working out a rigorous proof.

**Conjecture 1.11.** *Reducing a snowballing* HEARS *clause will produce a parallel structure whose asymptotic speed is the same as the speed of the original structure.*

We can now state this rule in English as follows: "If a HEARS clause snowballs then reduce it", and more formally as follows:

*rule* *REDUCE-HEARS* (**) **TRANSFORM**

**:'PROCESSORS $PNAME_{PDV}$ *PENUMER* ... **if** *COND*1 **then**
**HEARS** $PNAME_{HBV}$ *HENUMER* ...'

$\wedge$ *PENUMER*.**CLASS** = *ENUMERS*
$\wedge$ *HENUMER*.**CLASS** = *ENUMERS*
$\wedge$ *COND*1.**CLASS** = *PREDICATE*
$\wedge$ *COND*2.**CLASS** = *PREDICATE*
$\wedge$ *COND*1.*FREE-VARS* $\subseteq$ *PDV*
$\wedge$ *COND*2.*FREE-VARS* $\subseteq$ *PDV*
$\wedge$ *SET*1=(**BOUNDBY** *PDV* (1) *HBV HENUMER*)
$\wedge$ *SET*1*a*=(**BOUNDBY** *PDV* (2) *HBV HENUMER*)
$\wedge$ *PROC*1=(**BOUNDBY** *PDV* (1) *PDV* $\emptyset$)
$\wedge$ *SET*2=(**BOUNDBY** *PDV NIL HBV HENUMER*)
$\wedge$ *PROC*2=(**BOUNDBY** *PDV NIL PDV* $\emptyset$)
$\wedge$ *PROCh*=(**BOUNDBY** *PDV NIL HEXPR* $\emptyset$)
$\wedge$ (**THEOREM**
$((SET1 \cap SET1a) \in \{\emptyset\ SET1\ SET1a\}$
$\wedge ((\emptyset \subset SET1 \subset SET1a \wedge COND1) \Rightarrow SET1 \cup PROC1 = SET2)$
$\wedge (COND1 \wedge COND2 \Leftrightarrow SET1 \cup PROCh = SET2)))$

$\rightarrow$

**:'PROCESSORS $PNAME_{PDV}$ *PENUMER* ...
**HEARS** $PNAME_{HEXPR}$ ... '

when this rule is applied to the current state, the bindings will be as follows:

**='PROCESSORS** $P_{l,m}, 1 \le m \le n, 1 \le l \le n-m+1$
...
**HEARS** $P_{l+k,m-k}, 1 \le k \le m-1$ ...'

*PNAME*='*P*'
*PDV*='$l, m$'
*PENUMER*='$1 \le m \le n, 1 \le l \le n-m+1$'
*HBV*='$l+k, m-k$'
*HENUMER*='$1 \le k \le m-1$'
*SET*1=$\{((l_1+k, m_1-k)) : 1 \le k \le m_1-1\}$
*SET*1*a*=$\{((l_2+k, m_2-k)) : 1 \le k \le m_2-1\}$
*PROC*1=$\{((l_1, m_1))\}$
*SET*2=$\{((l+k, m-k)) : 1 \le k \le m-1\}$
*PROC*2=$\{((l, m))\}$
*PROCh*=$\{((l+1, m-1))\}$
*HEXPR*='$((l+1, m-1))$'
*COND*1='$2 \le m \le n$'
*COND*2=**true**

**THEOREM** is a function whose argument is a symbolic set-theoretic expression whose atomic terms are set expressions. These expressions are principally created by the **BOUNDBY** function, whose

inputs are the bound variables list of the processor name id, an identity parameter, the form that defines the array references that comprise the array definition, and the enumerator (*if* any) for the array reference.

*PDV* will take values of sequences of bound variables, and *HBV* will be sequences of expressions.

*BOUNDBY* is a quaternary function which returns an object that acts like a set-valued expression with free variables. Its four arguments are:

- A sequence of variable names, called VNS.
- An identification index, called IDX.
- A sequence of expressions, called EXS.
- A set of enumeration operators, called EOPS.

I gave each argument a name here for easy reference. **BOUNDBY** composes the object to return by first associating a "subscripted" new free variable (not to be confused with an array element reference) with each variable in VNS. The subscript is IDX (and if IDX is *NIL* there is no subscript). Every occurrence of an element of VNS in EXS or EOPS is also subscripted.

In the **BOUNDBY** expression defining *PROCh*, *HEXPR* (implicitly existentially quantified) is constrained by the **THEOREM** and *PROCh*= ... expressions.

Some bounds on the range of possible values for *HEXPR* are necessary. Something like

$$\forall z\, \exists y{:}z \in HEXPR \Rightarrow z \in \{'y', 'y+1', 'y-1'\} \wedge y \in PDV$$

would serve.

*COND*2 is also constrained by the theorems that can be proven.

This rule reduces the **HEARS** clauses from the large **PROCESSORS** statement of the current state to

**HEARS** $P_{i,m-1}$
**HEARS** $P_{i+1,m-1}$

The resulting **PROCESSORS** statement is

**PROCESSORS** $P_{i,m}, 1 \le m \le n, 1 \le i \le n-m+1$
**HAS** $A_{i,m}$
if $m=1$ then **USES** $v_i$, **HEARS** Q
if $2 \le m \le n$ then
**USES** $A_{i,k}, 1 \le k \le m$
**USES** $A_{i+k,m-k}, 1 \le k \le m$
**HEARS** $P_{i,m-1}$
**HEARS** $P_{i+1,m-1}$

*Figure 5.* Final Form of Main Processors Statement in $P$-time Dynamic Programming Derivation

#### *1.3.2.2 Rule A5: Write the Individual Processors' Programs*

The general idea of the rule is that the first rule isolated the deepest enumeration in the specification which assigned a value to an array element, and built the beginnings of a parallel structure where

each array element within the domain of that enumeration had its own private processor. Since the enumeration in time has been replaced by an enumeration in space, the layers of enumeration that get us to the point which induces the creation of the first parallel structure can be stripped away.

A technical note is that the enumerations can only be completely discarded when there is no calculation at intermediate levels. If there is such calculation, the system will have to add it to the appropriate processors when it strips away the layers of enumeration that include such calculation as well as the deeper enumeration. This does not make the asymptotic behavior of the parallel structure any slower except when the calculations include enumerations. When this is the case, it might be possible to respecify the problem to have separate copies of the array enumerated in the calculation for each cell of the target array. This would require an array whose dimension is the sum of the dimensionalities of the two arrays.

This rule is expressed in English as follows: "Supply each processor specified by a **PROCESSORS** statement with a copy of those enumerations from the original program that occurred within the region that included the assignment to array elements that generated that **PROCESSORS** statement. The references to array elements are replaced by associative lookups from the table of information that the processor has **HEAR**d. The outer enumerations are stripped from the program, and uses of the variables that were bound in these outer enumerations are replaced by constants reflecting the processor's ID."

The derivation of the $P$-time dynamic programming parallel structure is almost complete. It remains only to reduce the depth of enumeration to the single level implicit in the segment,

$$A_{l,m} \leftarrow \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$$

Rule A5 does this. The complete parallel structure that results is as follows:

**ARRAY** $A_{l,m}, 1 \leq m \leq n, 1 \leq l \leq n-m+1$
**PROCESSORS** $P_{l,m}, 1 \leq m \leq n, 1 \leq l \leq n-m+1$
  **HAS** $A_{l,m}$
  **if** $m=1$ **then USES** $v_l$, **HEARS** Q
  **if** $2 \leq m \leq n$ **then USES** $A_{l,k}, 1 \leq k \leq m$
    **USES** $A_{l+k,m-k}, 1 \leq k \leq m$
    **HEARS** $P_{l,m-1}$
    **HEARS** $P_{l+1,m-1}$
**ARRAY** $v_l, 1 \leq l \leq n$ **INPUT**
**PROCESSORS** Q **HAS** $v_l, 1 \leq l \leq n$
**OUTPUT ARRAY** $o$
**PROCESSORS** R **HAS** $O$

(*include if* $m=1$): $A_{l,1} \leftarrow v_l$ $\quad \theta(1)$

(*include if* $m>1$): $A_{l,m} \leftarrow \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$ $\quad \theta(n)$

(*include if* $l=1 \wedge m=n$): $O \leftarrow A_{1,n}$ $\quad \theta(1)$

#### 1.3.2.3 Rule A6: Improve Topology of Input/Output

We discovered that the rules described so far will produce a parallel structure in which every processor is directly connected to the input and output processors when given a specification of array multiplication. Only one I/O processor is created per I/O array, and for many problems,

including array multiplication, it is necessary to get some input or output from/to every processor. ($P$-time dynamic programming is an exception, in which only $\theta(n)$ of the $\theta(n^2)$ processors receive input values and the output is only a single value.)

We therefore conceived another rule to attempt to reduce the excessive connectivity that results from every processor needing access to input or output.

This rule is also not yet formulated in V, but it states that if the following conditions are met:

- the number of processors $n_i$ in a family that receives input from or sends output to a given processor is asymptotically unacceptable, and
- there is a **HEARS** clause $H_0$ such that the number of processors that do not **HEAR** any processor using $H_0$ clause (if input) or that are not **HEAR**d by any processor using that clause (if output) is asymptotically less than $n_i$,

then the I/O **HEARS** clauses can be reduced so that only those processors at a source (or terminus if output) of $H_0$ are directly connected to the I/O processor.

#### 1.3.2.4 Rule A7: Create Interconnections in a Family to Reduce I/O Connectivity

Rule A6 allows the reduction of connections from/to an I/O processor where a set of interconnections already exists to solve the I/O-free portion of the problem. In some problems, including array multiplication, no convenient set of interconnections exists and one must be introduced solely to distribute I/O values. Fortunately, the rule that would do this is fairly simple to state and is evidently implementable, given the mechanisms already required for *REDUCE-HEARS*.

The rule is: where a single **USES** clause telescopes, order the induced partition (definition 1.9) by the processor indices and interconnect the processors in each partition with a new **HEARS** clause where each processor is connected (only) to its immediate predecessor (if any) in this ordering.

## §1.4 A Derivation of Fast, Parallel Array Multiplication

Computer scientists have proposed many parallel schema for the array multiplication problem, probably because it is a practically important problem and seems so obviously amenable to parallel processing. One of the prettiest parallel structures is described in [KungLei-76]. Kung's algorithm multiplies an $n \times n$ array in $\theta(n)$ time using $\theta(n^2)$ processors of constant size. (Kung makes the assumption that a solution that involves $\theta(n)$ processors in communication with the outside world is acceptable. This subsection follows that assumption.) The best known sequential algorithm uses $\theta(n^{2.81})$ multiplications, but the obvious parallel structure using $n^{1.81}$ processors to do the job in linear time does not work; processors have to wait for other processors and have to receive copies of their results.

With the rules and postulated mechanisms for deriving information not locally obtainable from the specifications it does not seem possible to derive Kung's systolic array. It is, however, possible to derive another parallel structure with linear execution time. We added rule A7 with this derivation in mind, but do not feel that A7 is contrived or impractical.

Our parallel structure is inferior to systolic arrays because it uses more processors on a restricted class of matrices called "band matrices," in which all but a narrow diagonal band of the input matrices (and therefore of the output matrices) contains zero values.

The starting point of this derivation is a specification of array multiplication (we are assuming square

arrays to simplify the discussion):

| | |
|---|---|
| INPUT ARRAY $A_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| INPUT ARRAY $B_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| ARRAY $C_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| OUTPUT ARRAY $D_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| ENUMERATE $i \in \{1 \ldots n\}$ | $\theta(1)$ |
| ENUMERATE $j \in \{1 \ldots n\}$ | $\theta(n)$ |
| $C_{i,j} \leftarrow \sum_{k \in \{1 \ldots n\}} A_{i,k} B_{k,j}$ | $\theta(n^3)$ |
| $D_{i,j} \leftarrow C_{i,j}$ | $\theta(n^2)$ |

The use of arrays $C$ and $D$ seems redundant, but its purpose is technical - our rules would not permit us to assign multiple processors to a single array if that array were an INPUT or OUTPUT array. Duplicating all of the arrays in this manner, to avoid all appearances of "prejudicing the case" of which array's parallelism would be important, would only change in the resulting parallel structure in that each processor would be replaced by a family of three.

*MAKE-PSs* and *MAKE-IOPSs* add PROCESSORS statements,

| | |
|---|---|
| INPUT ARRAY $A_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| PROCESSORS PA HAS $A_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| INPUT ARRAY $B_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| PROCESSORS PB HAS $B_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| ARRAY $C_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| PROCESSORS $PC_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ HAS $C_{l,m}$ | |
| OUTPUT ARRAY $D_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| PROCESSORS PD HAS $D_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ | |
| ENUMERATE $i \in \{1 \ldots n\}$ | $\theta(1)$ |
| ENUMERATE $j \in \{1 \ldots n\}$ | $\theta(n)$ |
| $C_{i,j} \leftarrow \sum_{k \in \{1 \ldots n\}} A_{i,k} B_{k,j}$ | $\theta(n^3)$ |
| $D_{i,j} \leftarrow C_{i,j}$ | $\theta(n^2)$ |

*MAKE-USES-HEARS* completes the rough form of these statements.

ARRAY $A_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ INPUT
PROCESSORS PA HAS $A_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
ARRAY $B_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ INPUT
PROCESSORS PB HAS $B_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
ARRAY $C_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
PROCESSORS $PC_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ HAS $C_{l,m}$
| USES $A_{l,k}, 1 \leq k \leq n$
| USES $B_{k,m}, 1 \leq k \leq n$
| HEARS PA
| HEARS PB
OUTPUT ARRAY $D_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
PROCESSORS PD HAS $D_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
| USES $C_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
| HEARS $PC_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
ENUMERATE $i \in \{1 \ldots n\}$ $\theta(1)$
  ENUMERATE $j \in \{1 \ldots n\}$ $\theta(n)$
    $C_{i,j} \leftarrow \sum_{k \in \{1 \ldots n\}} A_{i,k} B_{k,j}$ $\theta(n^3)$
    $D_{i,j} \leftarrow C_{i,j}$ $\theta(n^2)$

*REDUCE-HEARS* is unable to improve this parallel structure, because there are no interconnections among the PCs to improve. Rule A6 is also helpless, although the topology of the interconnection graph is too rich ($\theta(n^2)$ rather than the goal of $\theta(n)$). Rule A7 comes to the rescue. Adding the HEARS clauses allowed by A7 and by the USES clauses of PC produces:

ARRAY $A_{l,m}, 1 \le l \le n, 1 \le m \le n$ INPUT
PROCESSORS PA HAS $A_{l,m}, 1 \le l \le n, 1 \le m \le n$
ARRAY $B_{l,m}, 1 \le l \le n, 1 \le m \le n$ INPUT
PROCESSORS PB HAS $B_{l,m}, 1 \le l \le n, 1 \le m \le n$
ARRAY $C_{l,m}, 1 \le l \le n, 1 \le m \le n$
PROCESSORS $PC_{l,m}, 1 \le l \le n, 1 \le m \le n$ HAS $C_{l,m}$
    USES $A_{l,k}, 1 \le k \le n$
    USES $B_{k,m}, 1 \le k \le n$
    HEARS PA
    HEARS PB
    if $m > 1$ then HEARS $PC_{l,m-1}$
    if $l > 1$ then HEARS $PC_{l-1,m}$
OUTPUT ARRAY $D_{l,m}, 1 \le l \le n, 1 \le m \le n$
PROCESSORS PD HAS $D_{l,m}, 1 \le l \le n, 1 \le m \le n$
    USES $C_{l,m}, 1 \le l \le n, 1 \le m \le n$
    HEARS $PC_{l,m}, 1 \le l \le n, 1 \le m \le n$
ENUMERATE $i \in \{1 \ldots n\}$ $\theta(1)$
  ENUMERATE $j \in \{1 \ldots n\}$ $\theta(n)$
    $C_{i,j} \leftarrow \sum_{k \in \{1 \ldots n\}} A_{i,k} B_{k,j}$ $\theta(n^3)$
    $D_{i,j} \leftarrow C_{i,j}$ $\theta(n^2)$

Then rule A6 is applied twice, and rule A5 once, finishing the derivation.

ARRAY $A_{l,m}, 1 \le l \le n, 1 \le m \le n$ INPUT
PROCESSORS PA HAS $A_{l,m}, 1 \le l \le n, 1 \le m \le n$
ARRAY $B_{l,m}, 1 \le l \le n, 1 \le m \le n$ INPUT
PROCESSORS PB HAS $B_{l,m}, 1 \le l \le n, 1 \le m \le n$
ARRAY $C_{l,m}, 1 \le l \le n, 1 \le m \le n$
PROCESSORS $PC_{l,m}, 1 \le l \le n, 1 \le m \le n$ HAS $C_{l,m}$
    USES $A_{l,k}, 1 \le k \le n$
    USES $B_{k,m}, 1 \le k \le n$
    if $m=1$ then HEARS PA
    if $l=1$ then HEARS PB
    if $m > 1$ then HEARS $PC_{l,m-1}$
    if $l > 1$ then HEARS $PC_{l-1,m}$
OUTPUT ARRAY $D_{l,m}, 1 \le l \le n, 1 \le m \le n$
PROCESSORS PD HAS $D_{l,m}, 1 \le l \le n, 1 \le m \le n$
    USES $C_{l,m}, 1 \le l \le n, 1 \le m \le n$
    HEARS $PC_{l,m}, 1 \le l \le n, 1 \le m \le n$
    $C_{l,m} \leftarrow \sum_{k \in \{1 \ldots n\}} A_{l,k} B_{k,m}$ $\theta(n)$
    $D_{l,m} \leftarrow C_{l,m}$ $\theta(1)$

## §1.5 Creation of Virtual Arrays, Processor Aggregation

### 1.5.1 An Informal Description of the Techniques

Consider the enumeration in Subsection 1.3.2.2,

$$A_{l,m} \leftarrow \bigodot_{k \in \{1 \ldots m-1\}} F(A_{l,k}, A_{l+k,m-k})$$

There is an enumeration, but only over r-values. For this reason, use of separate processors will not be generated for the steps of the enumeration.

Now one can make a few changes to the specification in order to generate separate processors for the steps of the enumeration. (This will be motivated later.)

Generate the following *virtualization*, creating the array $A'$

$$\begin{array}{l}
\textbf{ARRAY } A'_{l,m,k}, 1 \leq m \leq n, 1 \leq l \leq n-m+1, 0 \leq k \leq \#\{1 \ldots m-1\} \\
A'_{l,m,0} \leftarrow base_{\bigodot} \\
\textbf{ENUMERATE } k \in \langle\langle 1 \ldots m-1 \rangle\rangle \textbf{ do} \\
\quad A'_{l,m,\langle\langle 1 \ldots m-1 \rangle\rangle^{-1}(k)} \leftarrow A'_{l,m,\langle\langle 1 \ldots m-1 \rangle\rangle^{-1}(k)-1} \bigodot F(A_{l,k}, A_{l+k,m-k})
\end{array}$$

This structure represents several changes:

- First, it introduces a new dimension to the main array for each level of enumeration performed to find a value for the old elements of the array.
- Second, the enumeration $k \in \{1 \ldots m-1\}$ into the enumeration $k \in \langle\langle 1 \ldots m-1 \rangle\rangle$ is changed. This is perfectly legitimate—the set enumeration does not *forbid* enumeration in a specified order. When we consider automating this process, however, we should remember that there are $m!$ ordered enumerations corresponding to a specific unordered one of length $m$. The best orderings to try will probably include the arrival orderings inferrable from HEARS and HAS clauses, and the "natural" orderings, i.e. numerical order and inverse numerical order (where numbers are involved).

  Of course, this only applies when the inner enumeration(s) enumerate over a set. When the enumerand is already a sequence, this step and the fifth are unnecessary.
- Third, the value $base_{\bigodot}$, the value of $\bigodot_{k \in \emptyset}$, is introduced.
- Fourth, the function $\langle\langle 1 \ldots m-1 \rangle\rangle^{-1}$, the inverse of $\langle\langle 1 \ldots m-1 \rangle\rangle$ considered as a function, is also introduced. This will in fact be a function whenever it can be shown that the sequence has no duplicate elements, which will certainly be the case where the sequence is simply an ordering of some set, and will often be the case otherwise.
- Fifth, the running totals implicit in the $\bigodot_{\langle set \rangle}$ notation are explicated.

For $P$-time dynamic programming virtualization is worse than useless. The extra processors serve no purpose, they need to communicate with each other, and their existence forces the data to arrive in a specific order. More sophisticated virtualization heuristics could produce a different virtualization and eventually a different parallel structure by choosing a different base case and enumeration order. This technique is not useful on this specification.

However, consider the case of linear array multiplication. Application of the seven rules produces the following parallel structure:

ARRAY $A_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ INPUT
PROCESSORS PA HAS $A_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
ARRAY $B_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ INPUT
PROCESSORS PB HAS $B_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
ARRAY $C_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
PROCESSORS $PC_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$ HAS $C_{l,m}$
    USES $A_{l,k}, 1 \leq k \leq n$
    USES $B_{k,m}, 1 \leq k \leq n$
    if $m$=1 then HEARS PA
    if $l$=1 then HEARS PB
    if $m > 1$ then HEARS $PC_{l,m-1}$
    if $l > 1$ then HEARS $PC_{l-1,m}$
OUTPUT ARRAY $D_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
PROCESSORS PD HAS $D_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
    USES $C_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
    HEARS $PC_{l,m}, 1 \leq l \leq n, 1 \leq m \leq n$
    $C_{l,m} \leftarrow \sum_{k \in \{1 \ldots n\}} A_{l,k} B_{k,m}$ $\theta(n)$
    $D_{l,m} \leftarrow C_{l,m}$ $\theta(1)$

The asymptotic behavior of this parallel structure seems to be the same as that for Kung's parallel structure [KungLei-76]. However, there can be an advantage of Kung's parallel structure over the simpler one. With multiply "band matrices", where $j-i < k_{0,0} \vee j-i > k_{1,0} \Rightarrow A_{i,j}=0$ and $j-i < k_{0,1} \vee j-i > k_{1,1} \Rightarrow A_{i,j}=0$, it is possible to use fewer processing elements. If $k_{1,0}-k_{0,0}+1=w_0$ and $k_{1,1}-k_{1,0}+1=w_1$, then it can be shown that only $(w_0+w_1)n$ of the $n^2$ processors of our parallel structure can have non-zero answers, and only that many processors have to be provided. With Kung's parallel structure, however, only $w_0 w_1$ processors have to be provided. The multiplication takes $\theta(n)$ time. (It is possible to use the $\theta((w_0+w_1)n)$ processors to multiply the band matrices in $\theta(w_0+w_1)$ time, but this parallel structure cannot be synthesized automatically using these techniques, and in any event the time/processors tradeoff offered by Kung's parallel structure may be desirable.)

The virtualization process, alone, is not enough to synthesize Kung's systolic arrays. Notice that the number of processors in the parallel structure that results from the obvious virtualization is $\theta(n^3)$. Partial sums of product array elements reside in different processors at different times. This feature makes some technique like virtualization necessary to separate the computation of partial products, but processors have to be grouped to prevent this processor count blowup. Another more difficult technique, *aggregation*, will reduce the processor count to the target level.

Heuristically, aggregation is the grouping together of processors, each of which does a small amount of work, into groups of processors, each represented by a single processor. Each processor does all of the work that any processor in the original group did, but this can still be done quickly because each of the processors in the original group had a small amount of work to do, and no two processors had to do their work at overlapping times.

The reason why Kung's parallel structure can multiply arrays in linear time using constant space per processor is that he has performed a virtualization on the summation of result array elements. He avoids the need for $n^3$ processors by a process called *processor aggregation*. Each processor is responsible for computing $\theta(n)$ elements of the virtual array.

Reasoning similar to that performed in the change-of-basis generator and theorem prover will serve us well here. The target interconnection structure is

**PROCESSORS** $P_{l,m}, -n \leq m \leq n, -n \leq l \leq n-m+1$
  **HAS** $C^l_{i,j,k}, 1 \leq i \leq n, 1 \leq j \leq n, 1 \leq k \leq n, i-j=l-m, k=\min(n-l+1, n+m+1, n)$
  **USES** $A_{i,j}, 1 \leq i \leq n, 1 \leq j \leq n, i-j=l$
  **USES** $B_{i,j}, 1 \leq i \leq n, 1 \leq j \leq n, i-j=m$
  **HEARS** $P_{l-1,m}$
  **HEARS** $P_{l,m+1}$
  **HEARS** $P_{l+1,m-1}$

which is Kung's structure. This requires two changes of basis of input arrays ($i-j$ of both $A$ and $B$, rather than either $i$ or $j$), and a change of basis for the $C$ array, as well as some rather subtle timing arguments and replacement of the summation of each $C$-array element over a set of integers to a summation over a sequence of integers.

The figures on the following pages illustrate the virtualization and aggregation processos, as they apply to an $n=3$ instance of a matrix multiplication problem.

### 1.5.2 Formal Definitions of Aggregation and Virtualization

**Definition 1.12.** *A virtualization of a parallel structure is a new parallel structure that results from*

- *adding a dimension to an array, say A, producing $A'$ as follows: if $A_{\vec{i}}$ is a defined element of A, and the computation of $A_{\vec{i}}$ is performed by enumerating n elements of some set or vector S and performing a binary operation on a running total and each element of S as it is enumerated, then $A'_{\vec{i}|m}$ for $0 \leq m \leq n$ will be a defined element of the new array, $A'$;*
- *making the enumeration of S an ordered one; and*
- *replacing the original enumeration/calculation with a calculation that explicitly folds the $j^{th}$ value of the ordered enumeration as performed for $A_{\vec{i}}$ by operating on $A'_{\vec{i}|j-1}$ and that $j^{th}$ element.*

*The process of creating a virtualization is also called virtualization.*

**Definition 1.13.** *An* aggregation *of a parallel structure is a new parallel structure that results from partitioning the old set of processors of a family into equivalence classes, and creating a processor for each equivalence class. A processor in the aggregation* **HEAR**s *another such processor if any processor in the first equivalence class* **HEAR**d *any processor in the second.*

*The process of creating an aggregation is also called aggregation.*

There are, of course, an intractible number of possible aggregations according to this definition. Only simple aggregations are worthy of consideraton, because allowing complex ones would lead to a combinatorial explosion and because the complex ones would tend either to leave too many interprocessor connections or to have too much work being done in some of the processors.

Suppose that the virtualized family of processors is defined as

$$P_{z_1,z_2,\ldots z_m}, (enumers) \text{ (written } P_{\vec{z}})$$

We feel that interesting aggregations would identify

$$\forall \vec{z}, l: \exists P_{\vec{z}}, P_{\vec{z}+l\vec{i}} \Rightarrow P_{\vec{z}} \equiv P_{\vec{z}+l\vec{i}}$$

where $\vec{i}=\langle(i_1, i_2 \ldots i_m)\rangle$, all the $i_j \in \{-1, 0, 1\}$, and l ranges over integers. Here $\exists P_{\ldots}$ means that a given processor exists (and is not out of bounds of the original virtualization.) Early aggregation systems will confine themselves to this case.

### 1.5.3 What Virtualization Can and Cannot Accomplish

An important measure of the cost of a parallel structure is the product of the number of processors, the size of each one, and the amount of time the parallel structure takes to do a calculation. I will call this the PST measure.

PST=$\theta((w_0 + w_1)n^2)$ for the simpler parallel structure for matrix multiplication, when applied to band matrices of widths $w_0$ and $w_1$. Virtualization and aggregation can improve this to $\theta(w_0 w_1 n)$ by reducing the number of processors while allowing the size of the processors and the running time of the algorithm to remain the same.

It is possible to achieve PST=$\theta((w_0 + w_1)^2 n^2)$ by other means. This is equivalent whenever $w_1=\theta(w_0)$. Divide the $n \times n$ array of potential processors into $(w_0 + w_1) \times (w_0 + w_1)$ blocks and introduce input and output connections at the appropriate edges of each such block. This is impossible to derive by techniques shown so far, or reasonable extensions to them. It has the further disadvantage that the number of connections to input and output processors is $\theta(n)$ , while the same number is $\theta(w_0 w_1)$ for the systolic array parallel structure that results from virtualization and aggregation. A complexity measure that took into account the connections to the I/O processors would favor the systolic array structure even over the improved simple matrix multiplication scheme.

It should be noted that the parallel structure resulting from partitioning the potential processors has the same PST as systolic arrays, but P and T are different. Different measures, such as $PST^2$ may make different parallel structures more desirable.

## §1.6 Practicality Considerations

In addition to the results described above, we have investigated the problems that will be encountered when automatically derived parallel structures are used. A parallel structure will in general specify a collection of interconnections that may not correspond to any "off the shelf" product. We have begun to develop several concepts which Kestrel intend to explore further in 1983, but we will describe them briefly here. These considerations will be important when actual use of a system for automatically generating parallel structures is contemplated.

### 1.6.1 Basis Change

The topology of a parallel structure may be the same as that of an existing multiprocessor machine, but this fact may not be evident because of the nature of the indices. Suppose, for example, that multiprocessor systems of various sizes organized as square grids were commonly available, but that a user had submitted an instance of $P$-time dynamic programming to the parallel structure generator and received the result described above. The parallel structure's topology fits half of a square grid, but this fact is "hidden" under our choice of indexing. A change of basis can expose this fit.

### 1.6.2 Granularity Considerations

Many of the rules in this derivation system (and most of the need for inference) results from our unwillingness to consider as realizable a parallel structure where every processor is connected to every other. A consideration we labelled *granularity* persuades us that even a parallel structure in which every processor is connected to only a constant number of other processors and where the interconnection diagram is planar may be unrealizable in the future, where it will be common to have more than one processor, but not a complete system, on a "chip".

The $d$-dimensional lattice architecture may not be the ideal architecture for hardware implementation for a couple of reasons to be discussed in this section. One reason is that the connections specified may be too rich for an efficient VLSI implementation.

When a multiprocessor system is built on a single chip, or when each processor of a multiprocessor system is on its own chip, the concepts we intend to introduce are of no importance. However, it is important to consider the case where each chip contains several processors, but not a complete system.

The maximum practical "pin count" of a chip may limit efforts to place ever increasing numbers of processors on a chip as our fabrication technology improves. This is a separate limitation from wire-count limitations and planarity limitations. For example, in a two dimensional array of processors (each processor has two coordinates, each within a range of integers, and $P_{i,j}$ is connected to $P_{i,j\pm1}$ and $P_{i\pm1,j}$) the interconnection is obviously planar and the number of wires is proportional to the number of processors. This topology is therefore realizable on a single chip or in a configuration with one chip per processor. However, if our technology would otherwise allow $N^2$ processors on a chip, and a system with $M \gg N^2$ processors is desired, the number of busses from one chip to others would be $4N$ (except for chips on the edges of the array). This may require more pins than can be placed on a compact package.

To see how the various proposed architectures fare under the criterion of minimizing pin count as processor-per-chip count increases, consider the following table.

| interconnection geometry | busses per N-processor chip in M-processor system |
|---|---|
| complete interconnection | $NM$ |
| perfect shuffle | $2N^*$ |
| binary hypercube | $N(\log(M/N))^*$ |
| $d$-dimensional lattice | $2dN^{(d-1)/d}$ |
| augmented tree | $2\log(N+1)+1$ |
| ordinary tree | 3 |

*Figure 6.* Interconnection Requirements for Various Architectures (tentative)

It may be possible to improve bounds marked with an * by an asymptotically small factor using suitable constructions. Such improvements will not yield a qualitative difference in the sense of the argument.

For any architecture above the horizontal line, any decrease in $\lambda$ (the element size of a chip's logic elements or integrated wires) is useless without a *proportional* decrease in the chip's pin spacing. This is not true for architectures below the line. For those architecture, it is possible to preserve the pin spacing as $\lambda$ decreases, provided the chip's area or pin density is increased modestly.

In the tree structured architectures, most of the processors will be in multiprocessor chips, which we call leaf chips because they contain the leaf processors. These chips each hold $2^j$ leaf processors for some $j$, plus $2^{j-1}$ other processors necessary to tie the leaves together. Pairs of chips, including leaf chips, will be tied together with single processor chips having three busses each (or five for augmented architectures; see Figure 7). The number of single processor chips is one less than the number of leaf chips.

A construction that eliminates the single-processor chips in return for increasing the buss connections required for all chips by a modest constant factor has been described [BhattLei-82].

# Acknowledgements to Section 1

I wish to take this opportunity to thank the various people who have been extremely helpful to me while I was preparing this paper:

- Tom Brown, who helped me to debug several of the more complex rules and who kept me honest as to what a theorem prover could be expected to do efficiently,
- Tom Pressburger and Cordell Green, who helped me to get the English right, especially in the description of Dynamic Programming, and
- the rest of the gang at Kestrel, who listened to innumerable versions of the talks I gave while I was debugging the concepts involved.

Section 2

# Inference Requirements Analysis and Implementation Proposal

by
Tom C. Brown
Kestrel Institute
October 1982

## §2.1 Introduction

Inference requirements for two of Richard King's concurrent computing system synthesis rules (*MAKE-USES-HEARS* AND *REDUCE-HEARS*) are analyzed and shown to be

- intractable in their more general forms
- tractable under realistic constraints which include the applications thus far considered.

The *ad-hoc* constraints bring to bear special-case decision procedures for extended Presburger arithmetic and systems of linear constraints [Shostak-77,79,81].

The first rule [§ 1.3.1.3] documents data-flow dependencies for iterative array computations. Each element of an $O(n^p)$ - element array is defined exactly once by a sequence of iterative array-element assignments using inputs and previously defined array elements. The solution is in effect a parameterized description of a disjoint covering of the computation-array index set. Under reasonable constraints this covering can be computed in linear time and verified (disjointness, completeness) in quadratic time, as a function of the number of iterated assignment statements in the input specification.

The second rule [§ 1.3.2.1] recognizes a phenomenon called *snowballing*, wherein each member of an $O(n)$-element ordered array or processor family depends on results of each predecessor. This $O(n^2)$ dependency pattern is reduced to an $O(n)$ connection pattern wherein each processor receives results from its immediate predecessor (or input) and forwards them (plus its own result) to its immediate successor (or output). Heuristic guidance for the solution is extracted from

- the physical adjacency postulate: processors with "nearby" indices are candidates for immediate connection (the *Hears* relation)
- the linearity postulate: each HEARS clause defines a linear one-dimensional ($O(n)$) subfamily of the processor index set.

These constraints are easily tested. Once verified, the snowballing property reduces to a simple test which, instead of being $O(2^{2^{2^n}})$ as in the general Presburger-arithmetic decision problem [Shostak-79] of which it is an instance, is *linear* (in the input HEARS-clause length, under reasonable assumptions, § 2.4-5).

## §2.2 Data Flow Analysis

The *MAKE-USES-HEARS* rule operates on a specification wherein a processor has been assigned to each computation array element and each I/O array. It extracts from the program a set of *inferred conditions* and corresponding USES and HEARS clauses. The conditions are inferred from index ranges of enumerated (iterated) assignment statements. The rule makes allowances for the fact that iteration index variables need not correspond to Processor index variables, or that first even and then odd rows may be computed, etc.

Consider the schema [King-82],

```
1  PROCESSORS P_{l,m}, 1 ≤ m ≤ n, 1 ≤ l ≤ n−m+1
2      HAS A_{l,m}
3  PROCESSORS Q
4      HAS v_l, 1 ≤ l ≤ n
5  PROCESSORS R
6      HAS O

7  enumerate l' ∈⟨⟨1 ... n⟩⟩ do
8      A_{l',1} ← v_{l'}

9  enumerate m' ∈⟨⟨2 ... n⟩⟩ do
10     enumerate l' ∈{1 ... n−m'+1} do
11         A_{l',m'} ← ⊙_{k'∈{1...m'−1}} F(A_{l',k'}, A_{l'+k',m'−k'})

12 O ← A1,n
```

Following line 2 we should use lines 7-8 to infer the condition

```
(P.3a) if m=1 then
           USES v_l, 1 ≤ l ≤ n
           HEARS Q
```

because the assignment (line 8) binds $m$ to 1 and sets $A_{l',1}=v_{l'}(l'=1 \ldots n)$.

Similarly, we should adjoin the clauses

```
(P.3b) if 2 ≤ m ≤ n then

1          USES A_{l,k}, 1 ≤ k ≤ m−1
           HEARS P_{l,k}, 1 ≤ k ≤ m−1
2          USES A_{l+k,m−k}, 1 ≤ k ≤ m−1
           HEARS P_{l+k,m−k}, 1 ≤ k ≤ m−1
```

*where again the inferred condition $2 \leq m \leq n$ is derived directly from the controlling enumeration* (line 9). The two subclauses (1) and (2) are not part of the inferred-conditions derivation whose automation is the subject of this section; however, the rule derives (1) by selecting $A_{l',k'}$, in line 11 and noting that the definition of $A_{l',m'}$, uses $A_{l',k'}$ for $k'=1, \ldots, m'-1$, and similarly for (2) using $A_{l'+k',m'-k'}$. These mechanisms are already encoded in King's rule.

In general the *inferred conditions* problem is, given declaration

$$\textbf{ARRAY } A_{\vec{i}}, i_1{:}R_1, \ldots, i_p{:}R_p \tag{1}$$

with domain $\{\vec{i}{:}R_1 \wedge \ldots \wedge R_p\}$ and a list of *iterated assignments*

$$\begin{array}{l}\textbf{enumerate } j_1{:}S_1\\ \quad\ldots\\ \quad\textbf{enumerate } j_q{:}S_q\\ \qquad A_{f(\vec{j})} \leftarrow G[A_{g_l(\vec{j},\vec{k}_l)}{:}1 \le l \le r]\end{array} \tag{2}$$

verify that the corresponding sets

$$\{f(\vec{j}){:}S_1 \wedge \ldots \wedge S_q\} \tag{2'}$$

form a *disjoint covering* of $\{\vec{i}{:}R_1 \wedge \ldots \wedge R_p\}$. Clearly this condition is best tested by expressing each condition (2') in the form

$$\{\vec{i}{:}S_1' \wedge \ldots \wedge S_q'\} \tag{3}$$

where $S_i' \Leftrightarrow \exists_{\vec{j}}.[f(\vec{j})=\vec{i} \wedge S_k(\vec{j})]$; moreover, (3) is exactly the inferred condition required. Clearly (3) is uniquely defined from (2') iff $f$ is injective (one to one) on $\{\vec{j}{:}S_1 \wedge \ldots \wedge S_q\}$; otherwise $A_{f(\vec{j})}$ is defined twice.

To ensure effectiveness of the reduction to (3) we require that $f$ be a *linear transformation* from $Z^q$ to $Z^p$:

$$f(\vec{j})_k = \vec{c}_k \times \vec{j} + d_k \tag{4}$$

where $\vec{c}_k \times \vec{j}$ is the inner product of $\vec{c}_k$ and $\vec{j}$. Similar linearity constraints are placed on $R_k$, $S_k$ – e.g. $R_k$ has the form

$$L_k \le C_k \times j_k + D_k \le U_k \tag{5}$$

where $L_k$, $U_k$, $C_k$, $D_k$ may contain $j_1, \ldots, j_{k-1}, n$ free.

Now the covering of $\{\vec{i}{:}R\}$ ($A$'s domain) is *disjoint* iff $S^I \wedge T^J$ is *unsatisfiable* for each pair $(S^I, T^J)$ such that $\{\vec{i}{:}S^I\}$ and $\{\vec{i}{:}T^J\}$ are distinct instances of (3). In this conjunction n is a Skolem constant. The disjointness condition can be readily tested if in (3),

$$S_1' \wedge \ldots \wedge S_q' \text{ is a Presburger formula with constants (e.g. } \vec{i}, n) \tag{6}$$

Then the decision procedure of [Shostak-79] applies. This condition is clearly satisfied by the above example, and all others in [King-82].

The covering condition can be tested similarly. If $\{\vec{i}{:}T_1\}, \ldots, \{\vec{i}{:}T_r\}$ are the instances of (3) then they cover $\{\vec{i}{:}R\}$ iff

$$\forall n, \vec{i}.[R \Rightarrow T_1 \vee \ldots \vee T_r]$$

which reduces to extended-Presburger decidability of

$$R \wedge\sim T_1 \wedge \ldots \wedge\sim T_r.$$

Notice that the **HEARS** clause for (2) is obtained by first transforming the assignment (2) with constraints $S_1 \wedge \ldots \wedge S_q$ on j into an assignment

$$A(\bar{i}) \leftarrow G^f[A_{g^l(\bar{i},k_l):1\leq l\leq r}]$$

with constraints $S_1^f \wedge \ldots \wedge S_q^f$ on $\bar{i}$. This implies that $g_l(\bar{j},\bar{k}_l)$ must also be linear. The $\bar{k}_l$ are variables bound by iterated operators in $G^f[\ldots]$ - e.g., $(\bigodot_{k'\in\{1\ldots m'-1\}} F(\ldots))$ in line 11 above.

To conclude, the *inferred-conditions* function requires moderate ability to reason about systems of symbolic inequalities in extended Presburger arithmetic, to rename variables and to invert linear operations appearing in such formulas.Initially the inferred-conditions transformation may be implemented (for the case considered) by an interactive flow-analysis with linear-operator manipulation and extended Presburger inference capabilities.

## §2.3 Reducing Processor Interconnection Degree

### 2.3.1 Problem Statement

Given a program statement

$$\textbf{PROCESSORS } PNAME_{PBV} \quad PITER \quad \ldots \quad \textbf{HEARS } PNAME_{HBV} \quad HITER \qquad (1)$$

where

$PBV$ = processor bound-variable list,
$HBV$ = $HBV(PBV,k)$,
$k$ = bound variable(s) not in $PBV$ iterated by:
$HITER$ = $HITER(PBV,n,k)$, iterator over $k$

Define $F=F^{(n)}=\{PBV:PITER(PBV,n)\}$, the *processor-family* (index set) and $H=H^{(n)}=\{(a,b):PITER(a,n) \wedge b=HBV(a,k) \wedge HITER(a,n,k)\}$ the *Hears* relation of (1). Define $H_a$, the processors **HEARS** by $a$.

$$H_a=\{b:H_{ab}\}$$

Recall now the definitions of "telescopes" and "snowballs":

▸ H *telescopes* if either $H_a \subseteq H_b$, $H_b \subseteq H_a$, or $H_a \cap H_b = \emptyset$, i.e. , $\forall a,b \in F.H_a \cap H_b \in \{\emptyset, H_a, H_b\}$

▸ H *snowballs* if it telescopes and $\forall a,b,z \in F.[\emptyset \subset H_a \subset H_b \wedge H_a \cup \{z\} = H_b] \Rightarrow [z=a]$*

If H snowballs then $HITER(PBV,n,k)$ in (1) is "reduced" by setting $k=k_0$ where $HITER(PBV,n,k_0)$ and $HBV(PBV,k_0)$ is the index in $H_{PBV}$ "closest" to $PBV$ (using sum of absolute coordinate-differences as metric).

2.3.2 Example:

An application of *MAKE-USES-HEARS* in [King-82] generates a statement

**PROCESSORS** $P_{l,m,}$ $1 \leq m \leq n$, $1 \leq l \leq n-m+1$
...
**If** $2 \leq m \leq n$ **then**
...
**HEARS** $P_{l,k}$, $1 \leq k \leq m-1$ (a)
**HEARS** $P_{l+k,m-k}$, $1 \leq k \leq m-1$ (b) (2)

Clauses (a) and (b) each generate snowballing Hears-relations, and are reduced respectively to

...
**HEARS** $P_{l,m-1}$ (a)
**HEARS** $P_{l+1,m-1}$ (b)

It may be helpful to illustrate the resulting pattern for the case $n=5$ (b):

*See the note at the end of this Section.

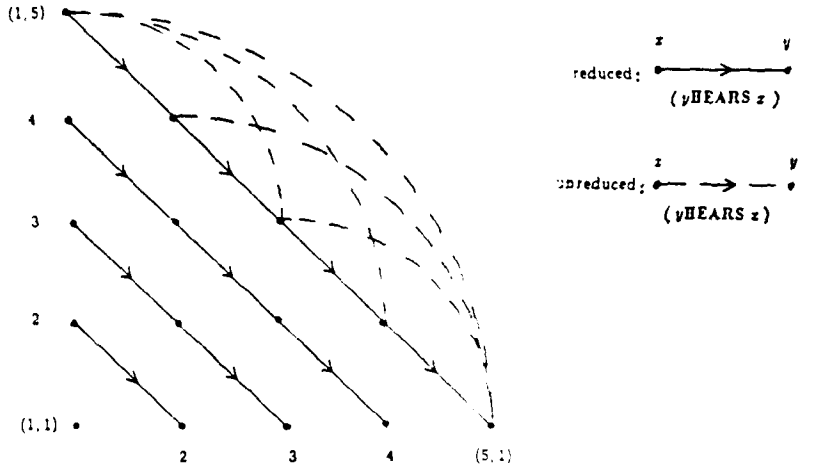



*Figure 7.* **HEARS** *clause* (2b)

Notice that in (a) the clause is reduced by setting $k=m-1$ whereas in (b) it is reduced by setting $k=1$. Both clauses can be effectively normalized so that the solution will be to set $k=$maximal value (below).

### 2.3.3 Remarks on "General Theorem-Proving Approach"

Without constraints on (1) the snowballing property can be quite intractable. Even if $PITER$, $HBV$, and $HITER$ are constrained so that only extended-Presburger formulas result, the problem may be intractable without additional constraints and/or expertise on the Presburger problem-domain.

Thus given (2b), we would extend a Presburger arithmetic basis (or specialized prover) with pairing axioms

$$hd\langle x,y\rangle=x, \quad tl\langle x,y\rangle=y$$
$$Integer(z) \vee \langle hd(z),tl(z)\rangle=z$$

Then

$$F(u) \Leftrightarrow 1 \leq tl(u) \leq n \wedge 1 \leq hd(u) \leq n-tl(u)+1$$
$$H_{uv} \Leftrightarrow tl(v)=tl(u)+hd(u)-hd(v)$$
$$\wedge\ 1 \leq hd(v)-hd(u) \leq tl(u)-1 < n$$
$$\wedge\ 1 \leq hd(u) \leq n-tl(u)+1$$

are derived following (1).

To prove Telescopes (H) we assume not, for some $a,b,\overline{a},\overline{b} \in F(n)$, and derive a contradiction:

$$
\begin{array}{ll}
F(a) \wedge F(b) \wedge F(\bar{a}) \wedge F(\bar{b}) & \\
H_{ac} & \left.\begin{array}{l}\\ \\ \end{array}\right\} H_a \cap H_b \neq \emptyset \\
H_{bc} & \\
H_{a\bar{b}} \wedge \sim H_{b\bar{b}} & \} \; H_a \not\subseteq H_b \\
H_{b\bar{a}} \wedge \sim H_{a\bar{a}} & \} \; H_b \not\subseteq H_a \\
\hline
\textbf{false} & \} \text{ via Integer-Arithmetic, Pairing Axioms}
\end{array}
$$

To prove Snowballs (H), we assert its negation for some $a, b, c, d$ in $F(n)$:

$$
\begin{array}{l}
F(a) \wedge F(b) \wedge F(c) \wedge F(d) \\
\sim H_{xz_1} \vee H_{yz_1} \vee \sim H_{yz_2} \vee H_{xz_2} \vee \sim H_{xz_3} \vee \sim H_{yz_3} \\
H_{ad} \\
\sim H_{ax} \vee H_{bx} \\
\sim H_{bd} \\
\sim H_{bx} \vee (H_{ax}) \vee [x{=}c] \\
a \neq c \\
\hline
\textbf{false}
\end{array}
$$

*where the second clause asserts "snowballs" and* $x, y, z_1, z_2, z_3$ are universally quantified variables. These axioms are in a form which can be given to the LMA prover [OverLusk-80].

We expect that specialized knowledge of extended-Presburger arithmetic decision procedures and integer programming will be required (at least) for success of so direct an approach to this class of problems. Another approach is to further constrain the problem without excluding the common cases of interest.

### 2.3.4 Heuristic Constraints

Notice that snowballing **HEARS** clauses define "one-dimensional" transitive relations over F – e.g. , the two-dimensional **HEARS** clause

$$\textbf{HEARS}\ P_{l',m'},\ l \leq l' \leq l + (m - m')$$

which "merges" (a) and (b) of (2) does not satisfy the "snowballs" predicate. Indeed its "reduction" would result in $O(n^2)$ processors sending data through two asymptotically hot wires. Thus we lose no generality in constraining $HITER$ (1) to iterate a single parameter (k) over a finite integer subrange dependent on $PBV$,n:

$$HITER = [L(PBV, n) \leq k \leq U(PBV, n)] \tag{3}$$

Another plausible constraint is that each one-dimensional subfamily of a snowballing **HEARS** be a "linear" subset of the lattice points over which $HBV$ ranges – e.g. , for $PBV$ fixed,

$$HBV(PBV, k) \text{ is linear in } k \tag{4}$$

Equivalently, the *first differential* in k

$$HBV(PBV, k+1) - HBV(PBV, k) \tag{5}$$

is independent of k. Indeed, we find plausible the stronger constraint,

$$(5) \text{ is constant (independent of both } k \text{ and } PBV). \tag{6}$$

After all, if (5) varies with $PBV$ then *distinct* colinear H-subsets of F have different slopes and are "likely" (though not required) to intersect, in violation of the *telescopes* constraint.

These constraints yield a *normal form* for each "linear snowball" (Figure 8):

$$\textbf{HEARS PNAME}_{F(z,n)+k\cdot C}, 0 \le k < L(z,n) \tag{7}$$

where C is a constant vector (the slope) and

$$z = F(z,n) + L(z,n) \cdot C \tag{8}$$

where $F(z,n)$ is the most-distant $HEARd$ point and $k = L(z,n) - 1$ selects the nearest $HEARd$ point (in taxicab metric: sum of absolute coordinate differences):

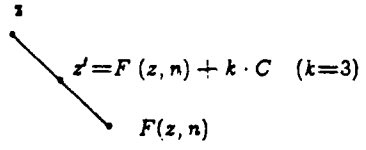



*Figure 8.* A Linear Snowball

Note that $F(z,n) = F(z',n)$ for each $z'$ on the line; thus $F(z,n) \neq F(z',n)$ implies $H_z \cap H'_z = \emptyset$.

**2.3.5** Example. The **HEARS** clauses of Example 2.3.2 have normal forms:

(a) **HEARS** $P_{(l,1)+k\cdot(0,1)}, 0 \le k < m-1$
(b) **HEARS** $P_{(l+m-1,1)+k\cdot(-1,1)}, 0 \le k < m-1$

**2.3.6** Linear Snowball Recognition-Reduction Procedure

Given **HEARS** clause (1) with $HITER$ as in (3):

*Step 1.* Verify (6)

*Step 2.* Put (1) in normal form (7)

*Step 3.* Verify (8)

*Step 4.* Verify (9) (for $0 < k \leq L(z,n)$):

$$F((F(z,n)+k\cdot C),n)=F(z,n) \tag{9}$$

*Step 5.* Reduce (7) to (10):

$$\textbf{HEARS}\ PNAME_{F(z,N)+(L(z,N)-1)\cdot C} \tag{10}$$

Failure of any verification attempt above implies return with failure (i.e. , the *REDUCE-HEAR* rule does not apply). This procedure suggests a refinement of King's rule to two rules, a *NORMALIZE-HEARS* rule which tests (6), and a *REDUCE-NORMALIZED-HEARS* rule which implements the remainder of this procedure. ∎

### 2.3.7 Correctness and Complexity of REDUCE-HEARS Refinement

The constraints (3)-(6) can be tested in linear time, provided that $HBV(PBV,k)$ contains no non-linear symbolic expressions in (PBV,k). (Note that a linearity claim must exclude perverse specifications such as $T(n)\times PBV(1)^2\times k^3$ where $T(n)$ is some arbitrary arithmetic formula which eventually simplifies to zero.)

Given similar non-perverse linearity constraints on $L(PBV,n), U(PBV,n)$ of (3), we assert linearity of the normal-form conversion (7). Condition (8) is a consistency test; it distinguishes the linear snowball $F(z,n)+k\cdot C$ from the non-snowballing **HEARS** index $F(z,n)+k\cdot C+D, D\neq\vec{0}$. Certainly it is conceivable that $F(z,n)$ and $L(z,n)$ might contain symbolic constants whose values would decide truth or falsity of (8); in this event *REDUCE-NORMALIZED-HEARS* should admit failure and ask the user what is going on. (Thus far we have no experience with such specifications).

Now (9) is precisely what we need (given (8)) to assert that H telescopes:

$$H_z\cap H'_z=\emptyset \Leftrightarrow F(z,n)\neq F(z',n);$$

$$H_z\cap H'_z\in\{H_z,H'_z\} \Leftrightarrow F(z,n)=F(z,n).$$

Again its verification (under the non-perversity assumption) requires only a linear-time simplification of a symbolic linear expression; the constraint that $k\leq L(z,n)$ has nothing to do with its truth or falsity.

To conclude, the *snowballs* antecedant (2) now reduces to

$$\{F(a,n)+k\cdot C : 0\leq k<L(a,n)\}\cup\{z\}$$

$$=\{F(b,n)+k\cdot C : 0\leq k<L(b,n)\},$$

which implies $L(b,n)=L(a,n)+1$ by telescoping ($F(a,n)=F(b,n)$). Therefore

$$z=F(a,n)+L(a,n)\cdot C=a$$

by (8), as required. We have proved the following:

**Theorem 2.1.** *If Procedure 2.3.6 returns successfully with reduced HEARS clause (10) then it is a reduction of the (linear) snowballing HEARS clause (1).* ∎

## §2.4 Conclusions

Significantly, Procedure 2.3.6 does recognize the class of snowballs thus far encountered (and which we expect to encounter) in *linear* time, instead of the super-exponential (worst-case) time which we might initially fear for the unconstrained theorem-proving approach of § 2.3.3. Both this and the *inferred conditions* problem illustrate the important heuristic of restricting the problem domain so that simple procedures can be applied.

# Note

The *REDUCE-HEARS* analysis is based on a somewhat less refined (and earlier) definition of "snowballs" than the one used in Section 1. Under the heuristic constraint of § 2.3.4 the two concepts are equivalent. R. King provided a discriminating example:

$$F = \{0, 1, \ldots, n\}$$
$$H = \{(l, k) : 0 \leq k < 2\left\lfloor \frac{l}{2} \right\rfloor \wedge l \leq n\}$$

It snowballs according to Section 2 but not according to Section 1. It violates the heuristic constraints of § 2.3.4 because $2\lfloor l/2 \rfloor$ is not a linear function of $l$. That it can be made into a snowball according to Section 2 by adjoining $n/2$ additional **HEARS** edges ("rounding and reducing") suggests that both definitions merit consideration. A sequel to this report will present a simplified analysis in terms of the more refined definition.

# Acknowledgements to Section 2

I wish to thank Richard King for his critical analysis, examples, counterexamples, and generally patient explanations of the workings of the synthesis rules. Both Richard and Carol Lei provided extensive assistance in the editing and TEXification of this document.

# References


[AhoUll-72] Aho and Ullman "The Theory of Parsing, Translation and Compiling"; Volume 1 *Pp. 314-320*

[AHU-74] Aho, Hopcroft and Ullman "The Design and Analysis of Computer Algorithms" *Pp. 67-68*

[BhattLei-82] Sandeep N. Bhatt and Charles E. Leiserson "How to Assemble Tree Machines" *Proceedings of the 14th Symposium on Theory of Computing, Pp. 77-88*

[Browning-80] Sally A. Browning "The Tree Machine: A Highly Concurrent Computing Environment" *California Institute of Technology Ph. D. Thesis*

[GCP-81] Cordell Green, Daniel Chapiro, and Thomas Pressburger "Research on Synthesis of Concurrent Computing Systems" *Kestrel Tech Report*

[GKT-79] L. J. Guibas, H. T. Kung and C. D. Thompson "Direct VLSI Implementation of Combinatorial Algorithms" *Procedings of the Caltech Conference on VLSI, January 1979*

[OverLusk-80] R. A. Overbeek and E. L. Lusk "Data Structures and Control Architecture for the Implementation of Theorem-proving Programs" LNC587, 5th Conference on Automated Deduction, 1980 *Pp. 232-249*

[King-82] Richard M King "Research on Synthesis of Concurrent Computing Systems" Tech Report KES.L 82.1, Kestrel Institute, May 1982

[Knuth-73] Donald Knuth "The Art of Computer Programming"; Volume 3 *Pp. 433-447*

[KungLei-76] H. T. Kung and Charles E. Leiserson "Systolic Arrays for VLSI"

[Shostak-77] Robert E. Shostak "On the SUP-INF Method for Proving Pressburger Formulas" JACM24 Oct. 1977 *Pp. 529-543*

[Shostak-79] Robert E Shostak "A Practical Decision Procedure for Arithmetic with Function Symbols" JACM26, April 1979 *Pp. 351-360*

[Shostak-81] Robert E Shostak "Deciding Linear Inequalities by Computing Loop Residues" JACM28, Oct. 1981 *Pp. 769-779*


DATE
ILME